Extended Instruction Exploration for Multiple-Issue Architectures

I-WEI WU and JEAN JYH-JIUN SHANN, National Chiao Tung University
WEI-CHUNG HSU, National Taiwan University
CHUNG-PING CHUNG, National Chiao Tung University

In order to satisfy the growing demand for high-performance computing in modern embedded devices, several architectural and microarchitectural enhancements have been implemented in processor architectures. Extended instruction (EI) is often used for architectural enhancement, while issuing multiple instructions is a common approach for microarchitectural enhancement. The impact of combining both of these approaches in the same design is not well understood. While previous studies have shown that EI can potentially improve performance in some applications on certain multiple-issue architectures, the algorithms used to identify EI for multiple-issue architectures yield only limited performance improvement. This is because not all arithmetic operations are suited for EI for multiple-issue architectures. To explore the full potential of EI for multiple-issue architectures, two important factors need to be considered: (1) the execution performance of an application is dominated by critical (located on the critical path) and highly resource-contentious (i.e., having a high probability of being delayed during execution due to hardware resource limitations) operations, and (2) an operation may become critical and/or highly resource contentious after some operations are added to the EI. This article presents an EI exploration algorithm for multiple-issue architectures that focuses on these two factors. Simulation results show that the proposed algorithm outperforms previously published algorithms.

Categories and Subject Descriptors: C [**Computer Systems Organization**]: GENERAL—*Instruction set design*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Extended instruction (EI), instruction set extension (ISE), multiple-issue architecture, customizable processor, application-specific instruction-set processor (ASIP)

ACM Reference Format:
I-Wei Wu, Jean Jyh-Jiun Shann, Wei-Chung Hsu, and Chung-Ping Chung. 2014. Extended instruction exploration for multiple-issue architectures. ACM Trans. Embedd. Comput. Syst. 13, 4, Article 92 (February 2014), 28 pages.
DOI: http://dx.doi.org/10.1145/2560039

1. INTRODUCTION

Next-generation digital entertainment and mobile communication devices will require higher processor performance. This can be achieved by increasing the clock rates. However, this approach has not been actively pursued in recent years due to its quadratic impact on power consumption. Another common approach is to extend the original instruction set architecture (ISA) with special instructions called extended instructions (EI) or instruction set extension (ISE) [Galuzzi and Bertels 2011; Pozzi and Ienne 2006] (note that an EI is an instruction in ISE). In this way, one instruction can be made

This research was financially supported by the National Science Council of the Republic of China under Contract No. NSC 98-2221-E-009-158-MY3.
Corresponding author's (I.-W. Wu) email: wuiw.tw@gmail.com.

DOI: http://dx.doi.org/10.1145/2560039

to perform the work of multiple operations. In many applications, the same operation patterns may be executed frequently. For example, multiply/accumulate (MAC) operation patterns are common in signal processing applications. The most straightforward approach is to extend a MAC instruction in the ISA and provide an accelerated functional unit in the implementation to execute it. In this article, a frequently executed operation pattern selected as a new instruction is termed an EI (extended instruction), and the dedicated (hardware) functional unit used to execute such an EI is known as an application-specific functional unit (ASFU). Other examples of ISEs/EIs are SSE, AVX, and NEON; these all extend the computation width rather than the computation depth. In addition to improving general execution performance, EI can be used to accelerate special applications. For example, instructions to support CPU virtualization, such as AMD-V and Intel VT-x, have been introduced to enable more efficient virtualization. Furthermore, EI can also reduce the binary code (or bytecode) size of a program because a single EI can represent multiple operations/instructions [David et al. 2001]. Many commercial extensible processors that have become available recently allow designers to exploit the power of EI. Some examples of these are Tensilica Xtensa [Halfhill 2003b], ARC ARCtangent [Halfhill 2000], MIPS CorExtend [Halfhill 2003a], ARM OptimoDE [Clark et al. 2004], and Altera Nios II [Altera 2004]. In this study, EI is used mainly to improve the execution performance of specific applications for embedded devices. In addition to increasing the clock rate and extending original ISA with EIs, multiple issue is a technique commonly implemented in superscalar and very long instruction word (VLIW) architectures. Like other microarchitectural techniques, it has gradually moved from general computing processors to embedded processors.

Several studies [Saghir et al. 2007; Atasu et al. 2003; Jain et al. 2004; Reddy 2006; Lü et al. 2008] have demonstrated that EI can be combined with multiple issuing to yield greater speedup in applications. Since multiple issuing is able to effectively exploit-instruction-level parallelism, the performance bottleneck may shift to the critical path of the application where the strength of EI lies. When a sequence of data-dependent operations are executed as one combined operation, its execution latency can be reduced by using ASFU. On the other hand, when the exploitation of a newly defined EI reduces the critical path of program execution, multiple issuing becomes important for speeding up the processing of instructions in noncritical paths. EI and multiple issuing complement each other, and the combination of both techniques achieves a critical balance for a high-performance processor.

The process of exploring EI for multiple-issue architectures differs from the single-issue case. In a single-issue architecture, packing more operations into one EI can yield more performance improvement. However, this is not always true for multiple-issue architectures, because packing an operation that is not on a critical path or does not use critical resources into an EI may not yield faster execution and could result in wasted resources and an increased area cost. Therefore, there should be a way to determine which operations are worth exploring before exploring EI on a basic block. From analyzing the operation (instruction) scheduling results on a multiple-issue architecture, we found that two factors should be considered when deciding the usefulness of each operation. The first is that the execution performance of a multiple-issue architecture is dominated by critical or highly resource-contentious operations. An operation in the critical execution path is called a critical operation, and an operation is considered to be contentious if there is a high possibility of delay in its execution due to hardware resource contentions (e.g., when the number of original functional units or the number of register read/write ports are limited) apart from data-dependent constraints. Thus, if critical and/or contentious operations can be selected to develop EI, the performance is then likely to be improved. Although both operation types (critical and contentious) are important for EI exploration, only the critical type has been considered in some

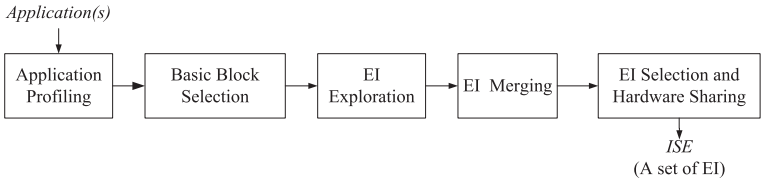


Fig. 1. EI design flow.

previous works [Clark et al. 2005; Lü et al. 2008]. The second factor is that these two properties (contentiousness and criticalness) may change when operations are replaced with new EIs, that is, an operation that was not critical or contentious may become critical and/or contentious after EI substitutions. However, if the EI exploration algorithm is not aware of such changes, opportunities to pack more critical/contentious operations into the EI may be lost, and the potential performance gain may not be achieved. This important factor has not been addressed in the current literature.

This article proposes an EI exploration algorithm for multiple-issue architectures that addresses these two factors. The proposed algorithm aims to minimize the execution cycle of the target application(s). To demonstrate the importance of the two factors, we compared our proposed algorithm against algorithms proposed in Lü et al. [2008] in terms of the speedup, area cost, and area efficiency (i.e., the speedup ratio per unit of the area cost). A study by Lü et al. [2008] derived EIs in a multiple-issue architecture. However, this study only considered the criticalness of the operations and not the contentiousness. We also compared different versions of the proposed algorithm, including variations that do not consider the critical and/or contentious properties of operations, or do not account for changes in the criticalness and/or contentiousness of operations. From these comparisons, the importance of the aforementioned factors can then be verified. The main contributions of this study are as follows.

(1) We provide an explanation of how current EI exploration algorithms do not adequately explore EIs for multiple-issue architectures.
(2) We describe the important factors for designing an EI exploration algorithm for multiple-issue architectures.
(3) We present proposal of an EI exploration algorithm for multiple-issue architectures that considers the important factors just described.
(4) We quantify the improvement resulting from the proposed approach in terms of the speedup, area cost, and area efficiency.

The rest of this article is organized as follows: Section 2 discusses the background for this work and previous related studies. Section 3 describes the proposed algorithm. Section 4 presents and discusses the simulation results. Conclusions are then presented in Section 5.

2. BACKGROUND AND RELATED WORKS

This section provides an overview of the EI design flow and discusses related studies. A motivating example is presented to illustrate the impact of the two important factors emphasized in this article.

2.1. EI Design Flow

The EI design flow is depicted in Figure 1. Once application profiling is complete, the most time-consuming basic blocks are selected as the inputs for EI exploration. The EI exploration process selects legal operation patterns as EIs which must conform to EI exploration constraints. In EI merging, several isomorphic EIs that would be derived

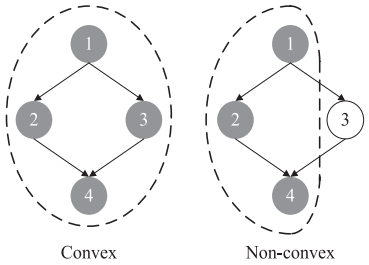


Fig. 2. The convex and nonconvex EIs.

in different basic blocks of the application are merged. For example, EI *B* is merged into EI *A* if EI *B* is a subgraph of or identical to EI *A*. After EI merging, EI selection sorts all derived EIs based on a predefined ordering, such as the speedup. Finally, some or all of the EIs are selected. Hardware sharing is also performed at this stage to achieve higher hardware utilization. Hardware sharing refers to the assignment of a hardware resource to multiple operations belonging to different EIs. To ensure that the performance is not reduced unnecessarily, the sharing process should be performed only if two EIs do not execute simultaneously. Note that hardware sharing is optional in the EI design flow. If it is assumed that the ASFU can concurrently execute multiple EIs, the hardware sharing process may be not required. In this article, we focus on EI exploration.

The constraints applied to EI exploration (I to V) and EI selection (II and V) are as follows.

(i) *Pipeline-Stage Timing*. Under this constraint, the execution time of ASFU must fit in a whole number of cycles of the original clock period of the processor core.
(ii) *ISA Format*. The ISA format imposes two constraints: the number of input/output operands employed by an EI and the number of EIs selected.
(iii) *Register File*. The number of input/output operands adopted by an EI cannot exceed the number of register file read/write ports.
(iv) *Convex Property*. The convex property is one where the EI output cannot be connected to its input set via operations not grouped in this EI to ensure feasible scheduling. Figure 2 shows examples of convex and nonconvex EIs (operations packed into an EI are circled with dashed lines).
(v) *Silicon Area*. This constraint restricts how much silicon area can be used for a single and/or all EIs.

2.2. Related Works

Several EI exploration algorithms have been proposed for single-issue architectures, and Galuzzi and Bertels [2011] provide a comprehensive survey of the topic. EI exploration algorithms can be classified into two categories. In the first type, the EI is grown from a set of fundamental operation patterns available in advance [Liem et al. 1994; Rao and Kurdahi 1992]. In the second type, no operation patterns are available in advance. All the EIs are then grown from an operation under EI exploration constraints. In the work proposed by Sun et al. [2002], EI exploration consists of two stages: EI enumeration and EI pruning. In EI enumeration, all operations that could be packed into EI are considered as basic EI candidates. By connecting several reachable basic EI candidates, larger EI candidates may be generated. After EI enumeration, all candidates are ranked by using a priority function that takes the number of inputs/outputs and the reduced cycles of the EI candidate into account. Then, the algorithm selects the

EI candidates whose ranking score is higher than a certain threshold ratio to be the final result. Furthermore, the exact algorithm proposed by Atasu et al. [2003] maps the EI search space, such as a basic block, to a search tree, where each level of the tree represents an operation in the input basic block. Except for leaf node, each node in the tree consists of two edges to its child node. One means to pack the operation into EI while another one does not. After building the search tree, the algorithm traverses the tree from the root to a leaf recursively and discards the path that violates EI exploration constraints. A path from the root to a leaf (i.e., an exploration result that consists of at least one EI) is regarded as a solution by EI exploration. However, as the exact algorithm is highly computationally intensive, it does not support a large search space. To overcome this problem, Pozzi et al. [2006] and Yu and Mitra [2007] proposed two algorithms that reduced the computational complexity of EI exploration. Pozzi et al.'s algorithm adopts a genetic algorithm where each solution is modeled by a gene. Yu and Mitra's algorithm recursively combines upward- or downward-connected subgraphs generated by different operations to form an EI. EI exploration also can be considered to be a hardware/software partition problem that divides all operations within a basic block into two groups: software (operations that will be executed on the original ALU) and hardware (operations that will be executed on the ASFU) [Biswas et al. 2006; Clark et al. 2005; Wu et al. 2007; Lü et al. 2008]. Similar to the hardware/software partition approach, most EI exploration algorithms provide a cost function to determine which operations should be packed into EIs.

To understand the effect of EI on multiple-issue architectures, AutoTIE [Goodwin and Petkov 2003] (note that EI is called fusion operation in AutoTIE) and Jain et al. [2004] applied an EI exploration algorithm [Atasu et al. 2003] to a very long instruction word (VLIW) processor. Although the algorithm they used was not designed for multiple-issue architectures (i.e., it does not consider the two important factors addressed in this article), it still achieved significant speedup. Adopting the same EI exploration algorithm [Atasu et al. 2003], Reddy [2006] proposed a fully automated design methodology for exploring the design space of a VLIW-application-specific instruction set processor (ASIP). His experimental results were in agreement with those of Jain et al. [2004] and demonstrated that EI can improve the performance of multiple-issue architectures. In Saghir et al. [2007], the authors manually identified operations for incorporating into the EI, including bit-reverse, bit-scrambling, bit-puncturing, and sum-of-absolute-differences operations, and implemented them on a soft VLIW processor. Their experimental results show that the augmentation of EI on a VLIW processor can deliver significant speedups using a small amount of FPGA resources. In contrast to these works, a study by Lü et al. [2008] was designed to automatically explore EI for multiple-issue architectures. In their work, the goal of exploring EI for multiple-issue architectures was to minimize the length of the critical path in the input basic block. They modeled the problem of EI exploration as a *covering minimal length* problem and solved it using dynamic programming. However, the work of Lü et al. only packs the critical operation (not including contentious one) into EI and overlooks the fact that the contentiousness and criticalness of an operation may change after EI substitutions.

Several processor architectures that aim to effectively combine EI with multiple-issue implementations have been proposed. Lx [Faraboschi et al. 2000] is a customizable multiclustered, very long instruction word (VLIW) processor. To support different customizations, Lx allows variations in the issue width, number of functional units (FU), functionality of each functional unit (FU), and instruction set architecture (ISA). Based on Lx ISA, ρ-VEX [Stephan et al. 2008] is a reconfigurable and extensible VLIW processor implemented using field-programmable gate arrays (FPGA). In both Lx and ρ-VEX, designers can customize the functionality of an FU to support EI. XiRisc, proposed by Lodi et al. [2003], consists of a VLIW processor and a reconfigurable unit

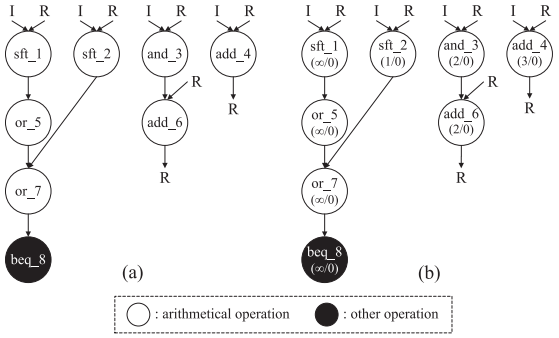


Fig. 3. The criticalness and the contentiousness of an operation (bitcount).

which is a programmable logic that can be dynamically configured to offer different functionalities for EI. The Xtensa processor [Halfhill 2003] is a commercial processor which allows designers to configure the issue width, register files as well as cache structures, and customize the functional unit for EI(s). Apart from the Xtensa processor [Goodwin and Petkov 2003], these works mainly focus on architecture design and not on customizing the functional unit for a specific application or algorithm. Therefore, in order to accelerate the development of customized functional units on these architectures, an algorithm that automatically generates EI is required.

2.3. Motivating Example

This section presents an example to illustrate the importance of the two factors mentioned in Section 1. Figure 3(a) shows a dataflow graph (DFG) that is transformed from one of the hot basic blocks in the benchmark Bitcount. A DFG of a basic block can be represented as a directed acyclic graph $D = (V, E)$. Each node $v \in V$ is an operation in the DFG. Each directed edge $(u, v) \in E$ represents the data dependency between two operations. The directed edge marked with R represents the values read from the register file. If the input value is an immediate value, the directed edge is marked with I. For example, operation 1 (sft_1) shifts a value (read from the register file) by the value in the immediate value field and writes the result to the register file. After scheduling operations in the DFG shown in Figure 3(a), the criticalness and the contentiousness of each operation can be found, as shown in Figure 3(b). In this article, the *criticalness* and *contentiousness* of an operation refer to whether the operation lies in the critical path and the possibility that it will be delayed during execution because of hardware resource limitations, respectively. The criticalness of infinite (∞) means that the operation locates on the critical path and the value of the criticalness will decrease as the path on which the operation locates becomes less critical. The contentiousness of an operation is proportional to the possibility of delaying its execution due to hardware resource limitations. In Figure 3(b), the criticalness and the contentiousness of each operation are given by the first and second numbers in parentheses within the node, respectively. An operation is critical if its criticalness is infinite (∞), while the contentiousness of zero represents that the operation will not delay its execution since there exists at least one functional unit that the operation can be scheduled on.

Based on the DFG shown in Figure 3, Figures 4(a), 4(b), and 4(c) depict the scheduling results for different EI exploration scenarios. Figure 4(a) does not account for the two important factors. It is labeled as 'Valid', since the exploration result is legal and attempts to pack as many operations into EI as possible. Figure 4(b) is based on Figure 4(a), with the difference being that criticalness and contentiousness have been considered. This is labeled as 'Static C/C'. Figure 4(c) shows the result when considering

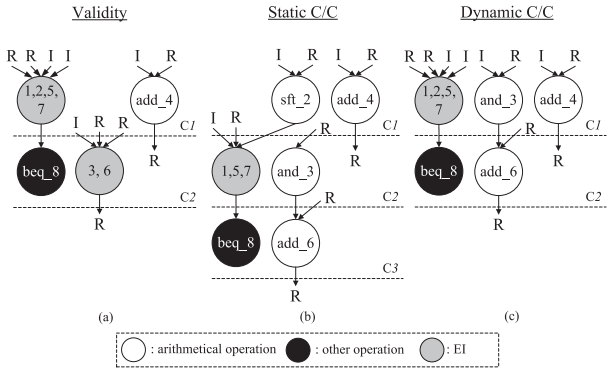


Fig. 4. Results of different EI exploration scenarios.

the two factors more aggressively. This is labeled as 'Dynamic C/C'. In Figure 4, we assume that (1) only arithmetic operations can be selected to form the EI; (2) the issue width and the number of original function units of the processor core are both three; (3) only one EI can be issued and executed at a time; (4) the number of register read/write ports is limited to six and three, respectively. In Figure 4, C_n represents the nth scheduling cycle. The scheduling result for Valid is shown in Figure 4(a). Since Valid does not consider the two factors, some operations that provide no benefit are selected (e.g., operations 3 and 6) to form the EI. Obviously, packing such operations in the EI does not improve the performance and instead wastes hardware resources. Figures 4(b) and 4(c) depict the scheduling results of Static C/C and Dynamic C/C, respectively. In both scenarios, an operation is selected to form an EI only if its criticalness is infinite or its contentiousness is larger than zero. However, Static C/C does not consider the second factor, which is the change in the criticalness and/or contentiousness of an operation. Thus, it does not account for the fact that operation 2 becomes critical after operations 1, 5, and 7 are replaced with the EI. As a result, Static C/C usually shows a much lower performance compared with Dynamic C/C.

3. EI EXPLORATION ALGORITHM

This section provides an overview of the proposed EI exploration algorithm and describes each major part of the algorithm in detail.

3.1. Overview

The proposed EI exploration algorithm consists of three major components: *operation profitability calculation*, *profitable operation packing*, and *illegal EI decomposition*. As mentioned earlier, the design of an EI exploration algorithm for multiple-issue architectures must account for two important factors: (1) only critical or contentious operations should be selected to form the EI, and (2) an operation's criticalness and contentiousness may change as a result of replacing operations with newly derived EIs. Therefore, the algorithm must be capable of (1) picking reachable critical or contentious operations to form the EI under EI exploration constraints and (2) identifying all critical and/or contentious operations iteratively.

Figure 5 depicts the flowchart of the proposed algorithm for EI exploration. The input and output are the hot basic block(s) selected from a target application and EIs, respectively. In Step 1, the algorithm calculates the criticalness and the contentiousness of each operation. Based on these results, the operation profitability (OP) of all the operations can then be determined. If an operation's criticalness is infinite or its contentiousness is larger than zero, then its OP is one; otherwise, the OP is zero. In

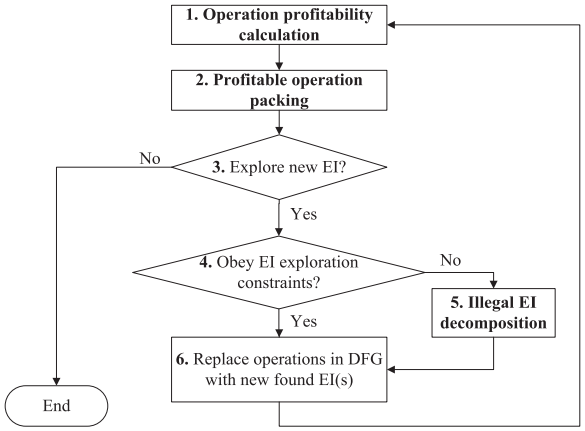


Fig. 5. EI exploration flow.

addition, if the OP of an operation is equal to one, it is considered to be a profitable operation and can be selected to form the EI. In Step 2, all reachable profitable operations are packed together to form the EI. Step 3 checks whether the termination condition is met. The termination condition is whether or not any new EI was derived. If no further EIs can be found, the algorithm terminates; otherwise, Step 4 checks the legality of each new EI. If any of the derived EIs violate exploration constraints, the algorithm will execute illegal EI decomposition (Step 5) to divide the illegal EI into several legal and smaller EIs; otherwise, it moves to Step 6. Finally, the algorithm replaces operations in the basic block with the newly discovered EI(s) (Step 6) and begins the next iteration from Step 1. Note that each newly discovered EI is considered to be a new operation in the basic block after operation replacement.

3.2. Operation Profitability Calculation

Two common approaches are used to accelerate the execution of an application. One approach is to reduce the length of the critical path; the other is to prevent operations from delaying execution due to hardware resource contentions apart from data-dependent constraints. In this article, we define criticalness and contentiousness to represent whether an operation locates on the critical path and the possibility that an operation may delay its execution because of the hardware resource contentions, respectively. Based on the criticalness and the contentiousness, the operation profitability (OP) of each operation can then be calculated. From the OP, the algorithm can determine which operation should be selected to form the EI.

The criticalness of an operation is the difference between its as-late-as-possible (ALAP) and as-soon-as-possible (ASAP) scheduling cycles, which are both determined by unconstrained ASAP and ALAP schedules. Here, 'unconstrained' means that no hardware resource limitation exists when scheduling operations. The ALAP and ASAP scheduling cycles of an operation i are denoted by $ALAP_i$ and $ASAP_i$, respectively. The criticalness of operation i is

$$Criticalness_i = \frac{1}{(ALAP_i - ASAP_i)}. \quad (1)$$

Operations located on the critical path have criticalness equal to infinity, and the criticalness will decrease as the path in which the operation is located becomes less critical.

Exploring EI must have the information of hardware resource usage. However, this hardware resource usage depends on the result of EI exploration. So when EI

exploration and operation scheduling are handled concurrently, it is hard to determine the hardware resource usage, since it will be affected by EI exploration. However, if EI exploration and scheduling are performed iteratively, the resource usage information could be determined by the compiler at every cycle. To resolve this problem, we define a variable, contentiousness, to represent the possibility that the execution of an operation will be delayed due to hardware resource contentions. For an operation, higher contentiousness implies higher possibility of delaying its execution, and the contentiousness of zero means that it would have at least one functional unit to schedule on or it is a critical operation. Accordingly, the contentiousness of an operation is defined as zero if its criticalness is infinite; otherwise, it can be calculated using the following steps. First, calculate $number\_of\_avaliable\_FU_i$, that is, the accumulation of the number of the available original functional units between the ASAP scheduling cycle to the ALAP scheduling cycle of operation i. If $number\_of\_avaliable\_FU_i$ is larger than one, it means there exists more than one functional unit on which operation i can be. Second, subtract one from the minimum of one and $number\_of\_avaliable\_FU_i$. The contentiousness of operation i is derived as follows.

$$Contentiousness_i = \begin{cases} 1 - Min(number\_of\_avaliable\_FU_i, 1), & \text{if } Criticality_i \neq \infty, \\ 0, & \text{if } Criticality_i = \infty. \end{cases} \tag{2}$$

$$number\_of\_avaliable\_FU_i = \sum_{t=ASAP_i}^{ALAP_i} \frac{number\_of\_avaliable\_oringial\_FU_t}{number\_of\_non\_critical\_operation_t}. \tag{3}$$

$$number\_of\_avaliable\_oringial\_FU_t = number\_of\_original\_FU - number\_of\_critical\_operation_t. \tag{4}$$

$$number\_of\_non\_critical\_operation_t = \sum_{i \in all\_non\_critical\_operation} \gamma \times operation_i. \tag{5}$$

$$\gamma = \begin{cases} 1, & \text{if } t \in ASAP_i \rightarrow ALAP_i, \\ 0, & \text{otherwise}, \end{cases} \tag{6}$$

where $number\_of\_non\_critical\_operation_t$ is the number of noncritical operations that can be scheduled in scheduling cycle t, $number\_of\_available\_original\_FU_t$ represents the number of original functional units available for noncritical operations (i.e., in which no critical operation is scheduled) at scheduling cycle t, $number\_of\_critical\_operation_t$ is the number of critical operations scheduled at scheduling cycle t, $number\_of\_original\_FU$ denotes the number of original functional units, and $ASAP_i \rightarrow ALAP_i$ represents all scheduling cycle of $operation_i$ between its ASAP scheduling cycle ($ASAP_i$) and its ALAP scheduling cycle ($ALAP_i$).

Figure 6 shows the contentiousness calculation step by step. The input DFG used in this example is shown in Figure 3(a). We assume that (1) only arithmetic operations are allowed to form an EI; (2) the issue width and the number of original function units of the processor core are both equal to three; (3) only one EI can be issued and executed at each scheduling cycle; (4) the number of register read and write ports is six and three, respectively. Initially, all operations are scheduled without any hardware resource constraints to identify their possible scheduling cycles. Figure 6(a) shows the possible scheduling cycle for all operations in the input DFG. In Figure 6(a), operations covered by a gray rectangular grid are critical, and only one scheduling cycle can be allocated to

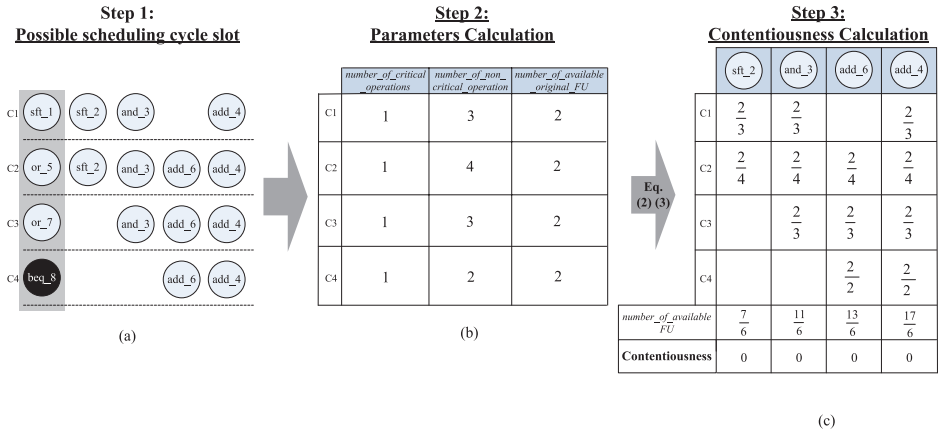


Fig. 6. Example of contentiousness calculation.

them; they are operation 1 at C1, 5 at C2, 7 at C3, and 8 at C4. Furthermore, since these operations are critical, their contentiousness defaults to zero according to Equation (2). In Step 2 (Figure 6(b)), *number_of_critical_operation*, *number_of_non_critical_operation*, and *number_of_available_original_FU* in all the scheduling cycles are calculated using Equations (4) to (6). Finally, in Step 3 (Figure 6(c)), using the results of Step 2, the contentiousness of all noncritical operations can be computed using Equations (2) and (3).

The OP of an operation is calculated from its criticalness and contentiousness. The OP of operation i is denoted as OP_i and is derived as

$$OP_i = \begin{cases} 1, & \text{if } Criticality_i = \infty \text{ or } C\text{ontentiousness}_i > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

3.3. Profitable Operation Packing

Profitable operation packing encapsulates a set of reachable profitable operations to form the EI. Note that not all profitable operations may be selected to form the EI. In this article, only the operations of integer arithmetic (except division), comparison, shift, and logic are permitted to form the EI. However, this limitation could be alleviated when increasing the implementation cost of ASFU. Furthermore, some of the packing results may be illegal (i.e., they may violate one of the constraints of EI exploration). The EI exploration algorithm will decompose an illegal EI into smaller legal ones by performing illegal EI decomposition. In addition, except for some multiple-cycle operations, the algorithm discards EIs that only contain a single operation, because such EIs cannot improve the performance. For multiple cycle operations, if ASFU could reduce their execution cycle, these operations would be permitted to form the EI even if only one operation exists in an EI.

3.4. Illegal EI Decomposition

The illegal EI decomposition process separates all of the operations in an illegal EI into several disjoint and legal subgraphs, where each subgraph is a set of reachable profitable operations that correspond to a legal EI. To maximize the speedup and minimize the area cost, the illegal EI decomposition tries to make all legal subgraphs (where each subgraph represents an EI) as large and as similar (isomorphic) as possible. Based on this objective, we defined a metric called *coverage* to quantify the decomposition result. The coverage of a legal subgraph denotes the sum of the sizes of all legal and fully/subidentical subgraphs. Legal subgraph X is considered to be fully/sub-identical to

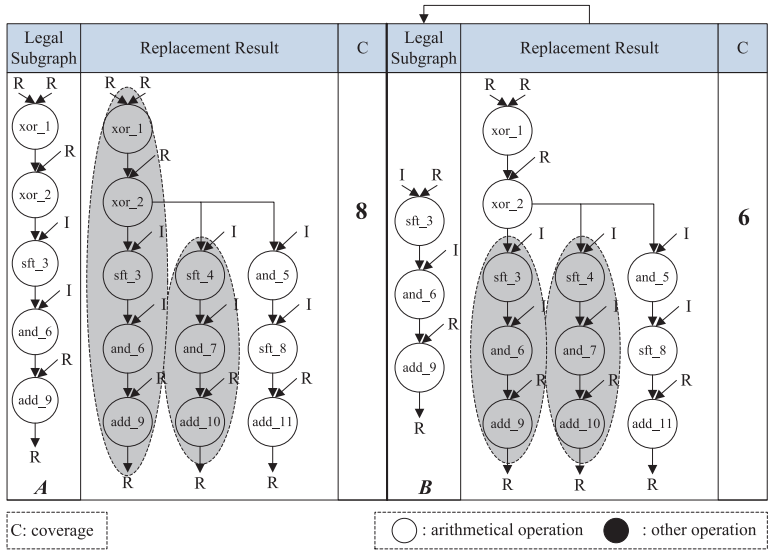


Fig. 7. Example of coverage.

legal subgraph Y only if the graph formed by X is isomorphic/sub-isomorphic with that formed by Y. Note that all subgraphs used to calculate the coverage must come from the same illegal EI.

A legal subgraph with a higher coverage has the advantages that (1) it can share its hardware resources with others (reducing the area cost), and (2) it can cover more operations in an illegal EI (delivering higher speedup). Figure 7 shows an example to illustrate the coverage calculation. Two legal subgraphs are depicted in the first and fourth columns of the table. The second and fifth columns of the table show the (operation) replacement results for legal subgraph A and B, respectively; the replacement results are circled with dashed lines. Based on the replacement result, the coverage (labeled as C in Figure 7) of each legal subgraph can then be determined. Since legal subgraph A could replace two groups of operations (operations 1, 2, 3, 6, and 9; operations 4, 7, and 10), its coverage is 8. On the other hand, two groups of operations are identical to legal subgraph B, and thus the coverage of legal subgraph B is 6.

Unfortunately, the problem of illegal EI decomposition is NP-complete, as can be proved through a trivial reduction to the graph-partitioning problem (by translating an illegal EI to a DFG)—a known NP-complete problem. Thus, in order to decompose an illegal EI in reasonable time, we derive a heuristic algorithm in this article. The basic idea of the proposed algorithm is to recursively merge several reachable operations to form an EI using a combination profit function under EI exploration constraints.

3.4.1. Flowchart of Illegal EI Decomposition. A flowchart of the illegal EI decomposition algorithm is depicted in Figure 8. The input and output of the decomposition algorithm are *operation_set*, which contains all operations in an illegal EI, and *EI_set*, which stores the decomposition result (i.e., decomposed legal EIs), respectively. Illegal EI decomposition involves two stages: *subgraph construction* and *subgraph combination*. First, subgraph construction generates upward and downward subgraphs for each operation in the illegal EI and stores the generated results in a set called *subgraph_set*.

The upward and downward subgraphs are each a sequence of operations starting from the operation or ending with it. For an operation, its corresponding upward

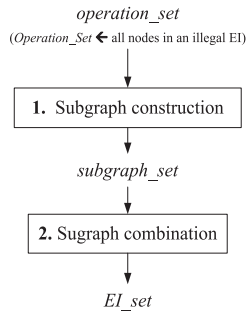


Fig. 8. Flowchart of illegal EI decomposition.

(downward) subgraph is generated by recursively packing its reachable predecessor (successor) operations with itself along one data-dependence chain until either the head (tail) operation of the illegal EI is reached or any of the EI exploration constraints are not conformed. Note that an operation usually consists of several upward and downward subgraphs. Moreover, each upward or downward subgraph must satisfy all EI exploration constraints. In the second step, subgraph combination computes the coverage for all subgraphs in *subgraph_set*. According to the coverage, subgraph combination iteratively merges reachable subgraphs under EI exploration constraints to form a larger subgraph. We will describe the subgraph construction and combination stages in detail in the following sections.

3.4.2. Subgraph Construction. Subgraph construction finds all legal upward and downward subgraphs for each operation in the illegal EI and stores them in *subgraph_set*. Figure 9 shows an example of a maximal upward and downward subgraph (a subgraph containing the maximum number of legal operations). In this example, the register file constraint is assumed to be 4/2 (read/write). Since no predecessor operation exists for operation 1, only one upward subgraph exists, that is, itself. Furthermore, since only one path exists between operation 3 and the head operation (operation 1), it also has only one maximal upward subgraph (the subgraph that includes operations 1, 2, and 3). On the other hand, the number of maximal downward subgraphs of operation 1 is three (operations 1, 2, 3, 6, and 9; operations 1, 2, 4, 7, and 10; operations 1, 2, 5, 8, and 11). Since only one path exists between operation 3 and its tail operation (operation 9), the number of maximal downward subgraphs is one (comprised of operations 3, 6, and 9).

3.4.3. Subgraph Combination. Subgraph combination iteratively merges connected subgraphs to form a bigger subgraph under EI exploration constraints. At this stage, all operations have corresponding upward and downward subgraphs. Many subgraphs in *subgraph_set* overlap with each other; in other words, an operation usually exists in several subgraphs. However, the output of illegal EI decomposition should be disjoint subgraphs. Thus, the subgraphs in *subgraph_set* cannot directly be considered as the outcome of illegal EI decomposition. Furthermore, in our experiment, the number of read/write ports of many subgraphs in *subgraph_set* is below the maximum allowed value. This implies that we can fuse multiple subgraphs to make a larger subgraph without violating EI exploration constraints (especially the register read/write constraint). Furthermore, in this work, except for multiplication, the execution cycle of the operation that can be packed into the EI is set to one. Therefore, subgraphs that contain only one single-cycle operation will be discarded after performing subgraph combination, since they cannot improve the performance.

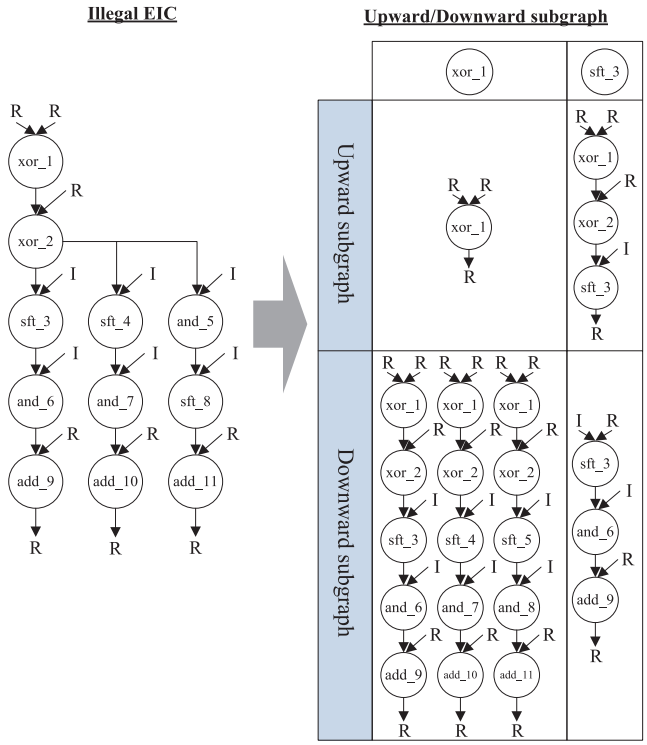


Fig. 9. Example of maximal upward and downward subgraphs.

Figure 10 shows the pseudocode of the four-step subgraph combination algorithm. Between Steps 1 to 3, the algorithm iteratively merges the subgraph with the largest coverage with others until no subgraph remains in *subgraph_set*. In Step 1, the algorithm calculates the coverage of all subgraphs in *subgraph_set* using CoverageCal() and then sorts the subgraphs in *subgraph_set* in decreasing order of coverage. If two subgraphs have the same coverage, the larger one (having more operations) has a higher priority and appears first in *subgraph_set*. In Step 2, based on the order in *subgraph_set*, the algorithm selects the subgraph with the highest combination profit, $subgraph_{highest}$. It merges as many reachable subgraphs as possible under EI exploration constraints. To represent the merged result, a data structure called $merged\_subgraph_{current}$ is introduced. Another data structure, $merged\_subgraph_{temp}$, is also created to substitute for $merged\_subgraph_{current}$. The function LegalityVerify() is used to verify the legality after merging a reachable subgraph with $merged\_subgraph_{current}$. If $merged\_subgraph_{temp}$ satisfies all EI exploration constraints and has better coverage than $merged\_subgraph_{current}$, $merged\_subgraph_{temp}$ becomes $merged\_subgraph_{current}$. Moreover, the algorithm removes all subgraphs that overlap with $merged\_subgraph_{current}$, that is, if a subgraph has any operation that exits in $merged\_subgraph_{current}$, it will be removed from *subgraph_set*. After examining all the subgraphs in *subgraph_set*, the algorithm then enters Step 3, which pushes the current merged result, $merged\_subgraph_{current}$, into *result_subgraph_set*. If *subgraph_set* has other subgraphs fully identical to $merged\_subgraph_{current}$, all these subgraphs are stored in *identical_subgraph_set* and removed from *subgraph_set*. Similarly, if a subgraph in *subgraph_set* overlaps with one in *identical_subgraph_set*, it is also removed from *subgraph_set*. Finally, the algorithm checks whether *Subgraph_set* is empty. If so,

```
Do
    /* Step 1: sort subgraph_set */
    For each subgraph_i of subgraph_set
        CoverageCal(subgraph_i, subgraph_set);
    End for
    Sort subgraph_set by coverage;
    /* Step 2: subgraph combination */
    subgraph_highest = pop subgraph_set;
    merged_subgraph_current += subgraph_highest;
    com_profit_current = CoverageCal(merged_subgraph_current, subgraph_set);
    merged_subgraph_temp = merged_subgraph_current;
    For each subgraph_i of subgraph_set
        If merged_subgraph_temp ∩ subgraph_i != Null
            merged_subgraph_temp += subgraph_i;
            com_profit_temp = CoverageCal(merged_subgraph_temp, subgraph_set);
            If LegalityVerify (merged_subgraph_temp) and com_profit_temp > com_profit_current
                merged_subgraph_current += subgraph_i;
                merged_subgraph_temp = merged_subgraph_current;
                Remove overlapped subgraphs with merged_subgraph_current from subgraph_set;
            End if
        End if
    End for
    /* Step 3: find fully/sub identical subgraph and remove overlapped ones */
    result_subgraph_set + = merged_subgraph_current;
    identical_subgraph_set = FindIndenticalSubgraph(merged_subgraph_current, subgraph_set);
    For each subgraph_i of identical_subgraph_set
        Remove overlapped subgraphs with subgraph_i from subgraph_set;
    End for
While (subgraph_set is not empty)
/* Step 4: remove the subgraph that contains only one single cycle operation */
For each subgraph_i of result_subgraph_set
    If subgraph_i contains one single cycle operation
        Remove subgraph_i from result_subgraph_set;
    End if
End for
```

Fig. 10. Pseudocode of the subgraph combination algorithm.

the algorithm terminates the subgraph combination loop and enters Step 4; otherwise, it continues to merge other subgraphs. In Step 4, the algorithm would discard subgraphs that consist of only one single-cycle operation, since these subgraphs cannot improve the performance. After removing valueless subgraphs, the algorithm outputs *result_subgraph_set* as the decomposition result (each subgraph in *result_subgraph_set* is an EI). Note that since some subgraph is discarded at Step 4, not all operations in the illegal EI are packed into EIs after the decomposition algorithm is terminated.

Figure 11 shows an example of illegal EI decomposition. In this example, the register file constraint is assumed to be 4/2 (read/write). The partition result is shown on the right side of Figure 11. In the first iteration, the subgraph (called *A*) containing operations *xor*, *xor*, *sft*, *and,* and *add* (i.e., 1, 2, 3, 6, and 9) is initially selected to perform subgraph combination, because it has the largest coverage of 8 (Step 1 in Figure 10). After merging with operations 4, 7, and 10, a new subgraph (called *A*') is formed. Since the subgraph containing the operations *and* and *sft* (5 and 8) can be legally merged with subgraph *A*', the algorithm merges them to form a new subgraph (called *A*") again. Then, since no other operation can be legally merged with *A*" further, the merging process (i.e., Step 2) is terminated. After subgraph merging, all subgraphs that are fully/sub-identical to subgraph *A*" must be removed. Therefore, operations 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 would be removed from the illegal EI. In the second iteration, since only

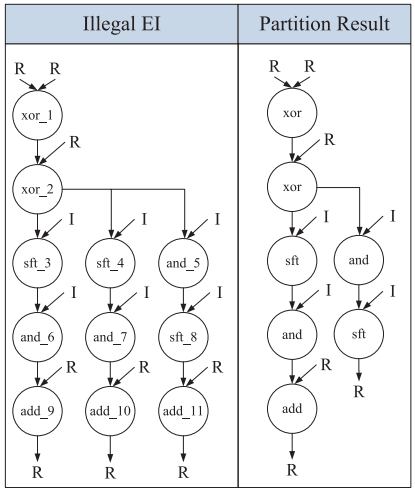


Fig. 11. Example of illegal EI decomposition result.

Table I. Characteristics of Selected Benchmarks

Name	# of selected basic blocks	# of operations in basic block		Instruction level parallelism (ILP)	
		Max	Avg.	Max	Avg.
ADPCM decode	25	13	3.60	4	1.40
ADPCM encode	24	13	3.54	4	1.39
Bitcount	18	27	5.66	8	1.92
Blowfish	7	430	71.86	20	3.52
CRC32	1	23	23.00	10	3.83
Dijkstra	6	12	5.33	7	1.29
Rijndael	12	903	112.92	65	5.62
Stringsearch	8	23	9.88	6	1.81

one operation (i.e., operation 11) exists, a subgraph (called B) containing operation 11 is formed. However, since subgraph B contains only one single-cycle operation, it would be discarded (Step 4). Accordingly, the decomposition result consists of only one subgraph, that is, subgraph A".

4. EXPERIMENTS AND EVALUATION

In this article, two EI exploration algorithms are compared in terms of the speedup, area cost, and area efficiency: the proposed algorithm and the one proposed by Lü et al. [2008]. Furthermore, to demonstrate the benefit of addressing two important factors (the criticalness and contentiousness of operations, and the change in these properties), the proposed algorithm was also evaluated in different scenarios. To examine the effect of the EI derived using the proposed algorithm in terms of the speedup of each benchmark, the structure of the EI and the associated speedups are presented in this section.

4.1. Experiment Settings

The benchmarks (ADPCM decode/encode, Bitcount, Blowfish, CRC32, Dijkstra, Rijndael, and Stringsearch) used in this experiment were selected from MiBench [Guthaus et al. 2001] and were compiled using LLVM 2.6 [Lattner 2002]. The characteristics of the selected benchmarks are listed in Table I.

Table II. Hardware Implementation Options Setting

Operation	Delay (ns)	Area (μm^2)	Operation	Delay (ns)	Area (μm^2)
Add/Sub	0.50	5768.0	Multiply	1.66	30814.6
AND	0.03	230.8	OR/XOR	0.03	242.7
NOR	0.02	173.1	Barrel shifter	0.26	4773.1
Compare	0.40	1204.2			

Table III. Area of the Register File Architectures (32×32-bit)

Spec.	Area (μm^2)	Number of single adders (Area/5767.978μm^2)
2R/1W	121,144.2	21.0
4R/2W	195,283.0	33.9
6R/3W	252,503.7	43.8
8R/4W	323,060.0	56.0
10R/5W	403,825.0	70.0

In this experiment, all the derived EIs were generated using the design flow shown in Figure 1. Note that EI selection is selecting how many explored EIs are to be realized in hardware. Since this experiment does not limit the number of EIs that can be realized in ASFU and the area cost of ASFU (i.e., no limitation on EI selection), all the derived EIs were considered to be the final results. Moreover, we made the following assumptions in this experiment.

(i) The execution frequency of the processor core is 300MHz ($\sim$3.3 nanoseconds). The base architecture is very long instruction word (VLIW), and all instructions are scheduled statically.
(ii) The software execution cycle of all the instructions is one cycle, except for multiplication (3 cycles) and division (12 cycles).
(iii) Several types of operations are prohibited from forming operation patterns. These include memory access, since it is difficult to estimate the execution cycle; flow control, to simplify the complexity of the control circuit of ASFU; and division, as well as floating point operations, due to the high cost per area of implementing these. However, other than memory operations, all prohibited operations in this work could be packed into EIs if a higher implementation cost of ASFU is granted. Since memory operations have nondeterministic latency, packing them into EIs may have negative impacts.
(iv) Only one ASFU exists. Thus, EI is allowed to execute when the hardware resource of ASFU is available.

In this experiment, hardware was synthesized using 0.13μm technology. All operations packed into ASFU are 32-bit. The synthesized results of the hardware implementation options are shown in Table II, and the area costs of various register files are shown in Table III. In this work, six different processor configurations were examined. Namely, two-issue with 4/2 and 6/3 register read/write ports (called 4/2/2 and 6/3/2 for short), three-issue with 6/3 and 8/4 register read/write ports (6/3/3 and 8/4/3), and four-issue with 8/4 and 10/5 register read/write ports (8/4/4 and 10/5/4). Among these configurations, 4/2/2, 6/3/3, and 8/4/4 are base configurations, while 6/3/2, 8/4/3, and 10/5/4 are extended from 4/2/2, 6/3/3, and 8/4/4, respectively (adding two/one register read/write ports). Accordingly, four register file settings were examined in this work. The proposed and related algorithms [Lü et al. 2008] were evaluated using an in-house-developed cycle-accurate VLIW simulator.

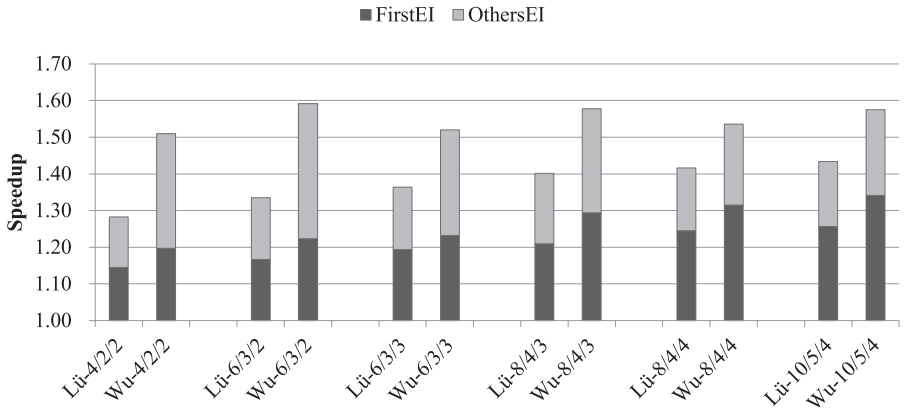


Fig. 12. Average speedup.

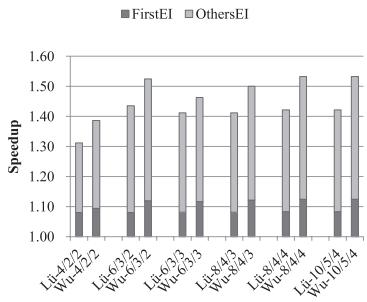
4.2. Experimental Results

In Figures 12 to 15, "Lü" and "Wu" denote the algorithms proposed by Lü et al. [2008] and the proposed algorithm, respectively; the three numbers after the dash represent the number of register read and write ports and the issue width. For example, Wu-4/2/2 indicates the proposed approach with 4/2 register read/write ports and a two-issue architecture. The results shown in Figures 12, 14, and 15 are averaged over all the benchmarks. Furthermore, since most of the performance gain was contributed by a particular EI in both approaches, all the experimental results are divided into two parts in the following figures. The first one shows a single EI that exhibits the highest speedup (called FirstEI), and the second includes all EIs other than FirstEI (called OthersEI).

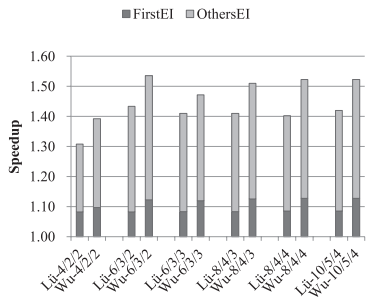
Figure 12 depicts the average speedup resulting from Lü et al.'s algorithm and the proposed algorithm. In all cases, the proposed algorithm yields better performance than Lü et al.'s algorithm. This is because the proposed algorithm iteratively selects critical and highly contentious operations to form EI, while Lü et al.'s algorithm only selects critical operations. In other words, as the EI derived from the proposed algorithm has more operations, it provides higher speedup. According to Figure 12, the speedup is mainly dominated by (1) the number of register read/write ports and (2) the issue width. Relaxing the register file constraint (i.e., increasing the number of register read/write ports) usually delivers better speedup. This is because relaxing the register file constraint allows the EI to legally contain more operations. The issue width is another important factor affecting the speedup. Since increasing the issue width would reduce the number of profitable operations, the size of each EI may be reduced, and thus the speedup achievable by EI may also be reduced.

Figure 13 shows the speedup of each benchmark with Lü et al.'s algorithm and the proposed algorithm. In most cases, the proposed algorithm yields higher speedup. In CRC32, Dijkstra, and four cases of Stringsearch (6/3/3 to 10/5/4), FirstEIs derived from both algorithms have the same speedup. For these benchmarks, the basic blocks where FirstEI is derived are the same in both algorithms, and all operations selected to form EI are critical operations (note: only in Stringsearch several operations are contentious operations when the issue width is two), and FirstEIs consequently derived by both algorithms are identical.

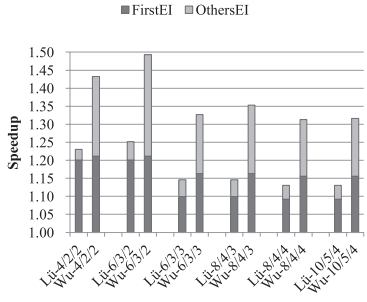
The average area cost that results from ASFU and the read/write port extension of the register file (Reg) is depicted in Figure 14. In this figure, each bar consists of several segments, where the bottom segment is the area cost used to extend the number



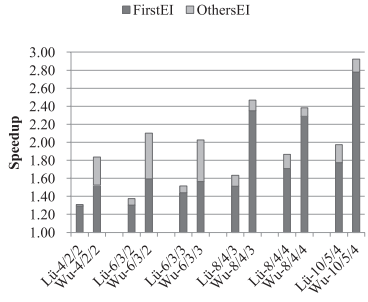
(a) ADPCM decode.



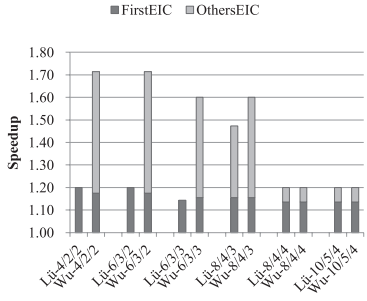
(b) ADPCM encode.



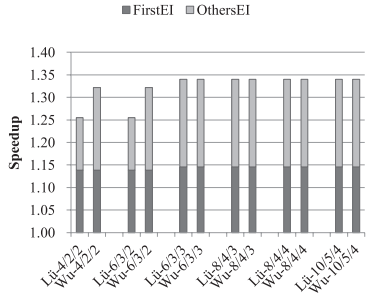
(c) Bitcount.



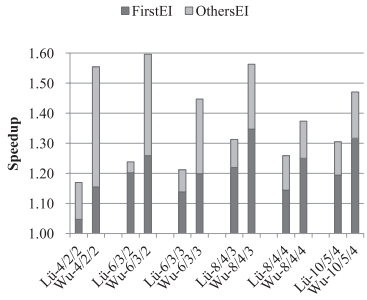
(d) Blowfish.



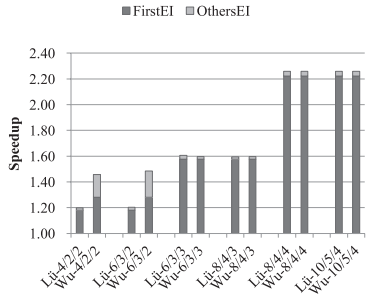
(e) CRC32.



(f) Dijkstra.



(g) Rijndael.



(h) Stringsearch.

Fig. 13. Speedup of individual benchmark.

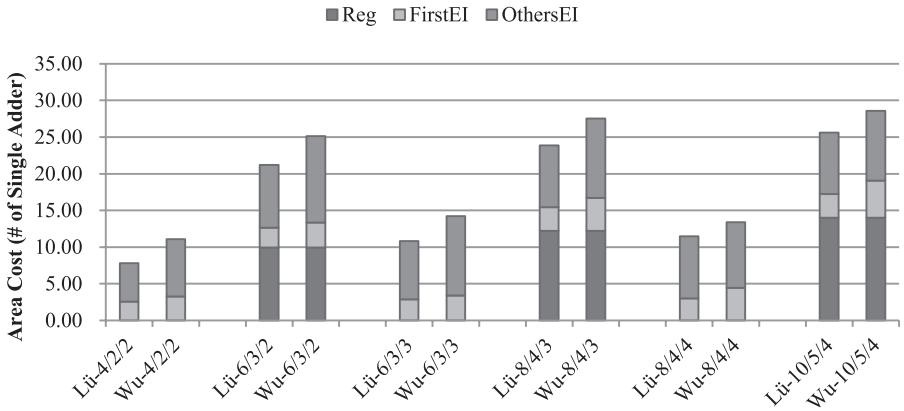


Fig. 14. Average area cost.

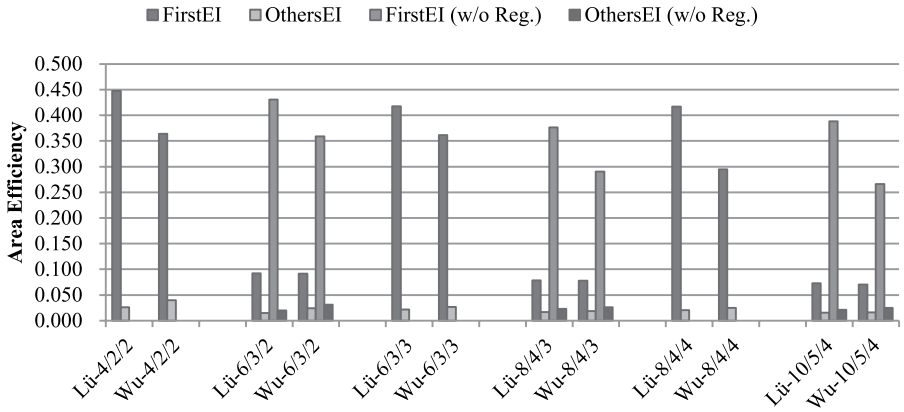


Fig. 15. Average area efficiency.

of register read/write ports (note: 4/2/2, 6/3/3, and 8/4/4 do not have this cost), and the other two segments indicate the area cost of FirstEI and OthersEI, respectively. The average area cost shown in Figure 14 is defined as follows.

$$\begin{cases} area\,cost_{First} = \dfrac{area\ size\ of\ EI_{First} + extra\ area\ size\ of\ register\ file}{area\ size\ of\ a\ single\ adder}, \\ area\ cost_{Others} = \dfrac{\sum_{i=2}^{n} area\ size\ of\ EI_i + extra\ area\ size\ of\ register\ file}{area\ size\ of\ a\ single\ adder}, \end{cases} \tag{8}$$

where *First* is FirstEI, *Others* represents OthersEI, n is the total number of EIs derived in each benchmark (EI_1 is FirstEI), EI_{First} EI_{First} is the FirstEI, *extra area size of register file* is the area cost used to extend the number of read/write ports of the register file, and the area of a single adder is $5768.0\mu m^2$. Note that since the architecture configurations of 4/2/2, 6/3/3, and 8/4/4 are considered the base configurations, no extra area cost of the register file is included (i.e., *extra area size of register file* is zero). However, the architecture configurations of 6/3/2, 8/4/3, and 10/5/4 are extended from the base configurations of 4/2/2, 6/3/3, and 8/4/4 by adding two/one register read/write ports, respectively. Therefore, these three configurations would introduce the extra area cost of the register file (i.e., *extra area size of register file* is not zero). In all the

cases, the proposed algorithm has a higher area cost than Lü et al.'s algorithm. Since Lü et al.'s algorithm only packs critical operations into EIs, the size of EIs is smaller than that of the proposed algorithm.

Figure 15 illustrates the average area efficiency with and without the extra area cost from the register file. The area efficiency is used to represent the speedup that can be achieved per unit area consumed by the ASFU and the register file (if needed), and is defined as follows.

$$\begin{cases} area\ efficiency_{First} = \dfrac{speedup_{First}}{areacost_{First}}, \\ area\ efficiency_{Others} = \dfrac{speedup_{Others}}{areacost_{Others}} \end{cases}, \tag{9}$$

where *First* is FirstEI, *Others* represents Others, $speedup_{First}$ and $speedup_{Others}$ are the speedup of FirstEI and OthersEI, respectively, and $area\ cost_{First}$ and $area\ cost_{Others}$ is from Eq. (8). Note that since the architecture configurations of 4/2/2, 6/3/3, and 8/4/4 do not incur the extra area size of the register file, the average area efficiencies with and without the extra area cost from the register file are identical. Therefore, each approach (Lü and Wu) only has two results (i.e., FirstEI and OthersEI) in these three configurations. In 6/3/2, 8/4/3, and 10/5/4, all the results are categorized into two categories: with and without the extra area cost of the register file. In most cases, Lü et al.'s algorithm has a higher area efficiency compared with the proposed algorithm. This is because it has a lower area cost compared with the proposed algorithm. Nevertheless, since the main objective behind adopting EI is to improve the speedup, the sacrifice of some area efficiency is acceptable. Furthermore, according to the results depicted in Figures 12 to 15, relaxation of the register file constraint may improve the performance but not the area efficiency.

Table IV depicts the execution time of Lü et al.'s algorithm and the proposed algorithm (platform: AMD Opteron 6172 (2.1GHz)). For the basic block size of each benchmark, please refer to Table I. Obviously, the execution time highly depends on the basic block size (i.e., number of operations in the basic block). For example, the execution time of benchmarks with larger basic block size, such as Blowfish and Rijndael, is significantly higher than others. Furthermore, in all cases, the proposed algorithm is slower than Lü et al.'s. This is because the proposed algorithm must iterate several times until no new EI are derived, while Lü et al.'s one does not. However, since EI exploration is performed statically, a longer execution time is acceptable if the better results can be achieved.

To show the impact of the two major factors considered in this study, we evaluated four of the scenarios based on the proposed algorithm that differ as follows: (1) in terms of the operation property (i.e., criticalness and contentiousness) that can be selected to form the EI, and (2) whether a change in the criticalness and/or contentiousness of operations is taken into account. Table IV shows the experimental results for these four scenarios. In this table, C_r, C_o, and L represent the criticalness, contentiousness, and all legal operations being selected to form the EI. M denotes that the exploration algorithm is aware of the change in criticalness and/or contentiousness of operation, while S denotes that it is not. In other words, the exploration algorithm is iteratively executed in M, while it is only executed once (single iteration) in S. The numbers following M or S denote the number of register read/write ports and the issue width.

Based on the results shown in Table V, the importance of two major factors considered in the proposed algorithm was demonstrated. First, the speedup resulting from the selection of both the critical and contentious operations to form the EI can be seen by comparing C_r-C_o-M and C_r-M. Second, if the algorithm considers that an operation may become critical and/or highly contentious after some operations are replaced with

Table IV. Execution Time of Lü et al.'s Algorithm and the Proposed Algorithm

Benchmark	Execution Time (millisecond)											
	Lü						Wu					
	4/2/2	6/3/2	6/3/3/	8/4/3	8/4/4	10/5/4	4/2/2	6/3/2	6/3/3/	8/4/3	8/4/4	10/5/4
ADPCM decode	20	12	12	11	12	11	32	15	15	16	15	15
ADPCM encode	20	11	11	11	11	11	31	14	14	14	15	15
Bitcount	52	17	16	16	16	17	80	21	21	21	24	24
Blowfish	28,595	26,473	24,332	24,349	24,399	24,399	66,843	98,999	50,565	57,653	97,099	67,366
CRC32	6	6	6	6	6	6	8	8	8	8	9	9
Dijkstra	4	4	4	4	4	4	5	5	6	6	6	6
Rijndael	173,272	162,996	149,042	121,967	157,493	108,306	361,975	311,492	312,039	181,461	180,496	410,876
Stringsearch	28	14	14	14	14	14	44	17	20	20	22	22

Table V. Averaged Results of Different Scenarios

	Speedup		Area cost (# of single adder)				Area Efficiency			
			FirstEI		OthersEI		FirstEI		OthersEI	
Scenario	FirstEI	OthersEI	w/o Reg.	w Reg.	w/o Reg.	w Reg.	w/o Reg.	w Reg.	w/o Reg.	w Reg.
C_r-C_o-M-4/2/2	1.20	0.31	3.29		11.09		0.364		0.040	
C_r-C_o-S-4/2/2	1.19	0.29	2.79		10.01		0.428		0.040	
C_r-M-4/2/2	1.16	0.22	2.92		10.50		0.400		0.029	
L-M-4/2/2	1.20	0.32	3.29		10.99		0.364		0.041	
C_r-C_o-M-6/3/2	1.22	0.37	3.41	13.33	15.20	21.72	0.359	0.092	0.031	0.017
C_r-C_o-S-6/3/2	1.20	0.33	2.91	12.83	14.36	21.38	0.414	0.094	0.029	0.015
C_r-M-6/3/2	1.19	0.21	3.03	13.59	14.38	20.63	0.394	0.092	0.019	0.010
L-M-6/3/2	1.22	0.37	3.41	13.33	15.83	22.35	0.358	0.091	0.029	0.016
C_r-C_o-M-6/3/3	1.23	0.29	3.41		14.23		0.362		0.027	
C_r-C_o-S-6/3/3	1.23	0.28	2.80		12.29		0.437		0.029	
C_r-M-6/3/3	1.19	0.20	2.98		14.02		0.400		0.018	
L-M-6/3/3	1.20	0.31	3.41		15.70		0.352		0.025	
C_r-C_o-M-8/4/3	1.29	0.28	4.46	16.69	15.29	23.07	0.290	0.078	0.026	0.012
C_r-C_o-S-8/4/3	1.29	0.27	3.86	16.09	15.45	23.83	0.334	0.080	0.023	0.011
C_r-M-8/4/3	1.20	0.23	3.67	15.90	14.18	22.74	0.327	0.076	0.021	0.010
L-M-8/4/3	1.27	0.31	4.46	16.69	17.89	25.67	0.284	0.076	0.023	0.012
C_r-C_o-M-8/4/4	1.32	0.22	4.46		14.64		0.295		0.025	
C_r-C_o-S-8/4/4	1.30	0.18	3.65		11.62		0.356		0.023	
C_r-M-8/4/4	1.22	0.22	3.67		13.43		0.331		0.023	
L-M-8/4/4	1.31	0.24	4.46		15.17		0.293		0.022	
C_r-C_o-M-10/5/4	1.34	0.23	5.05	19.05	15.07	24.02	0.266	0.070	0.025	0.010
C_r-C_o-S-10/5/4	1.31	0.19	4.24	18.24	12.70	22.47	0.310	0.072	0.023	0.009
C_r-M-10/5/4	1.25	0.28	4.14	18.14	14.79	24.65	0.302	0.069	0.027	0.012
L-M-10/5/4	1.30	0.27	4.93	18.93	15.97	25.05	0.264	0.069	0.024	0.011

newly discovered EI(s), then it will show a higher speedup. The C_r-C_o-M versus C_r-C_o-S example illustrates this. Furthermore, C_r-M exhibits poor speedup in benchmarks with a higher ILP, such as Blowfish, CRC32, and Rijndael. This is because an application with a higher ILP usually contains more contentious operations. Accordingly, if these contentious operations cannot be packed into an EI, the speedup would then be limited. Unlike C-M, C_r-C_o-S performs poorly in applications where operations change their property to profitable after the first iteration (Bitcount and ADPCM). Consequently, fewer operations are packed into the EI and fewer EIs are formed in C_r-C_o-S. This is why C-M and C_r-C_o-S have lower area cost. On the other hand, since several EIs derived by L-M contain some non-useful (i.e., nonprofitable) operations, L-M leads to the highest area cost in most cases.

4.3. Discussion of the Experimental Results of Scenario C_r-C_o-M

The results in the previous paragraph are averaged from all the benchmarks. To understand the performance impact of each benchmark in C_r-C_o-M, the following paragraphs discuss each benchmark individually. The discussion mainly focuses on two parts: (1) the dataflow graph (structure) of FirstEI derived under different hardware constraints (i.e. different microarchitecture configurations), and (2) the impact on speedup from FirstEI and OthersEI under different hardware constraints. Furthermore, to clearly identify the impact of individual constraints in terms of the speedup, when we

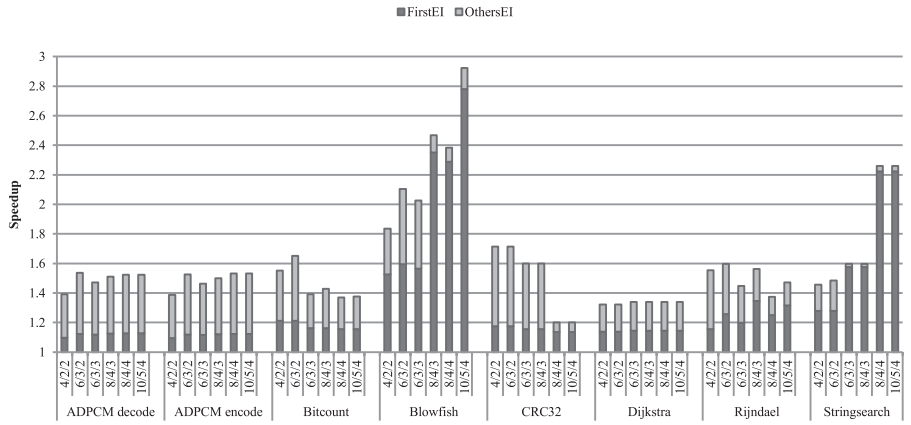


Fig. 16. Speedup of each benchmark in C_r-C_o-M.

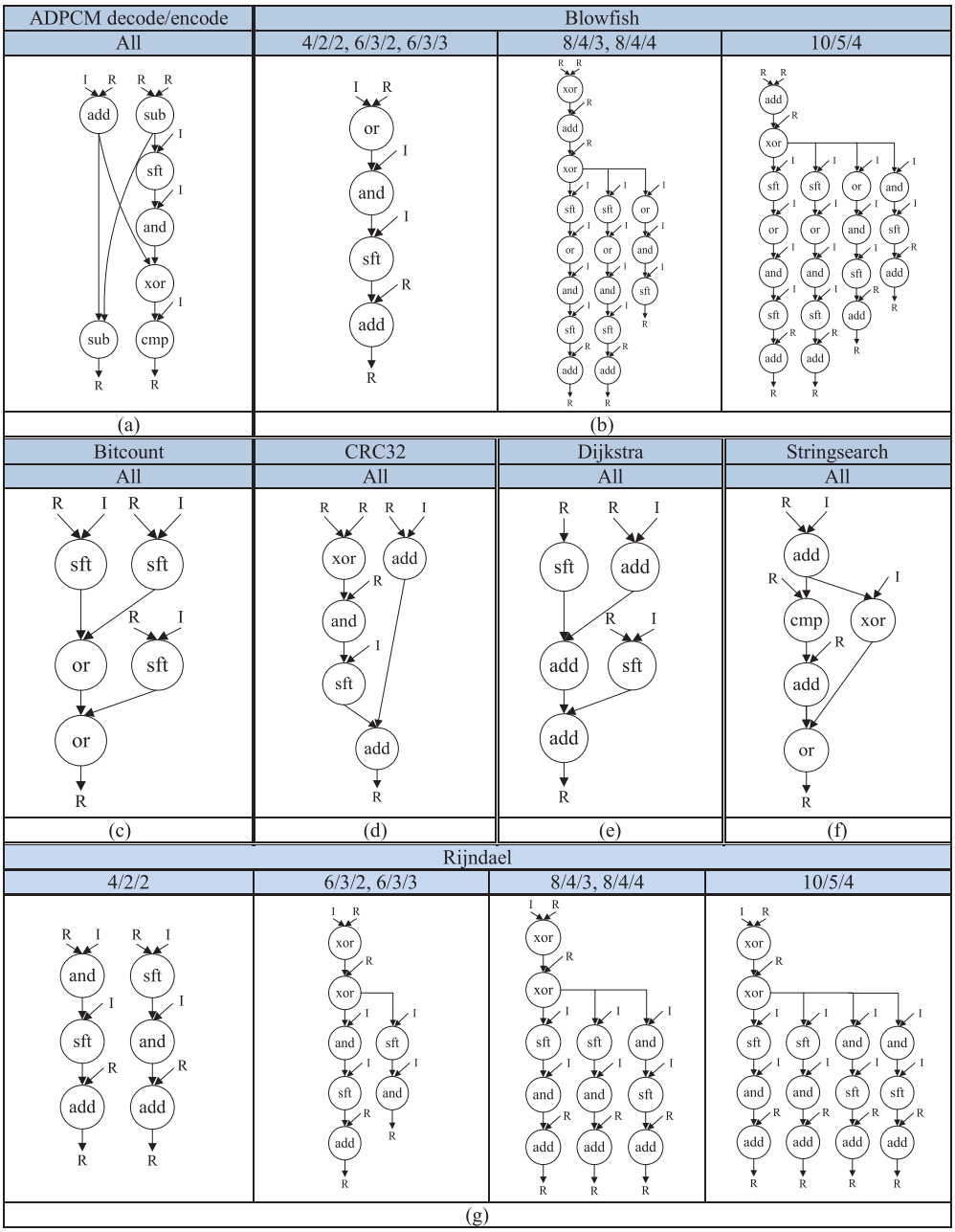
discuss the impact caused by relaxing the register file constraint, the issue width is fixed, and vice versa.

Figure 16 depicts the speedup of C_r-C_o-M for each benchmark with different hardware constraints. The dataflow graph of FirstEI, as derived from each benchmark, is shown in Table VI, where each number set listed above the data flow graph is the constraint of the register file and the issue width. The first and second numbers represent the number of register read and write ports, respectively, and the last number is the issue width. 'All' represents all hardware constraints used in this work, that is, from 4/2/2 to 10/5/4. In Table VI, "R" indicates that the input value of an operation originates from the register file, while "I" denotes an immediate value. The vertex is an operation, and each edge denotes the dependence between two operations. The type of operation is marked inside the vertex, where *cmp* and *sft* denote compare and shift operations, respectively.

ADPCM Decode and Encode. (Table VI(a)) ADPCM decode and ADPCM encode are two separate benchmarks. However, since the EIs derived in both benchmarks are almost identical, these two benchmarks are therefore discussed together. Under 4/2/2, all profitable operations in the basic block, from which FirstEI is derived, have been legally packed into the EI. Therefore, only one type of FirstEI is derived in ADPCM. The dataflow graph of FirstEI is shown in Table VI(a).

Relaxing the register file constraint from 4/2 to 6/3 (4/2/2 to 6/3/2) improves the performance of the processor core with an ASFU which realizes the functionality of FirstEI. (Note that the processor core with an ASFU realizing the functionality of the FirstEI and OthersEI is called Pro-FirstEI and Pro-OthersEI for short in the following discussion, respectively.) Under 4/2 (4/2/2), one operation in the basic block where FirstEI is derived may delay its execution because an insufficient number of register read ports are available. When the register file constraint becomes 6/3, this operation no longer delays execution and the performance improves. However, relaxing the register constraint further to 8/4 or more does not improve the performance. For Pro-OthersEI, relaxing the register file constraint generally increases the performance. When the register file constraint is 4/2, not all profitable operations can be packed into one EI. They need to be packed into multiple EIs. When the constraint is relaxed to 6/3, all profitable operations are packed into one EI, therefore improving the performance. In some cases, even though relaxed register file constraints do not lead to fewer packed EIs, more packed EIs may be able to execute in parallel with other operations, resulting

Table VI. Dataflow Graph of FirstEI Explored by C_r-C_o-M



in a performance increase. Therefore, relaxing the register file constraint from 6/3 to 8/4 still improves the overall performance. Relaxing the constraint further from 8/4 to 10/5 only marginally increases the performance. For this benchmark, having more than 8/4 read/write ports is of little benefit.

Increasing the issue width from 2 would provide little speedup for Pro-FirstEI. This is because (1) enlarging the issue width would reduce the total original execution

cycles (i.e., without EI), and (2) the number of profitable operations in FirstEI would not change. For Pro-OthersEI, enlarging the issue width leads to two different results. When the issue width increases from 2 to 3, several operations become noncontentious (i.e., nonprofitable), and the performance is reduced. However, when the issue width is increased to 4, no operation properties change, and the total original execution cycles are reduced. Thus, the performance will increase.

Bitcount (Table VI(c)). In Bitcount, only one kind of FirstEI is derived. This is because all profitable operations have been legally packed into FirstEI as a single EI under 4/2/2. The dataflow graph of FirstEI is shown in Table VI(c). In addition, it is notable that several EIs derived in Bitcount could be merged with FirstEI.

Relaxing the register file constraint does not improve Pro-FirstEI performance for two reasons. First, all profitable operations have already been legally packed. Second, none of the operations need to delay their execution because of insufficient register read/write ports. On the other hand, relaxing the register file constraint would improve the performance of Pro-OthersEI. When the register file constraint increases from 4/2 to 6/3, EI will legally contain more profitable operations and the performance will increase. On the other hand, when the register file constraint is relaxed from 6/3 to 8/4 or higher, more operations and EIs will be able to execute simultaneously (no more delays in execution), resulting in further performance gain.

Enlarging the issue width would reduce the performance of Pro-FirstEI. This is because increasing the issue width further would make several operations too unprofitable. This would reduce the number of EIs that can be merged with FirstEI. Consequently, the performance of Pro-FirstEI would then be reduced. For ASFU-OthersEI, enlarging the issue width would degrade performance because some operations would become unprofitable.

Blowfish (Table VI(b)). The EIs derived in Blowfish are highly dependent on the register file constraint. According to Table I, some basic blocks in Blowfish contain many operations and have a higher ILP. This implies that relaxing the register file constraint could potentially yield a larger EI. All FirstEIs derived in Blowfish are shown in Table VI(b). In addition, many EIs derived in Blowfish can also be merged with FirstEI, thereby increasing the performance potential of Pro-FirstEI.

Relaxing the register file constraint improves the performance of Pro-FirstEI, because it increases the size of the EIs being derived. For Pro-OthersEI, relaxing the register file constraint reduces performance except when 4/2 changes to 6/3. This is because when relaxing the register file constraint, several profitable operations have already been packed into FirstEI, and the size of the other EIs has decreased. As a result, the performance impact is lower than expected.

The performance of Pro-FirstEI would reduce with issue width increases. When increasing the issue width, a part of operations would become unprofitable. Consequently, fewer numbers of operations could be selected to form EI so that less performance would be achieved. Similar to Pro-FirstEI, the performance of Pro-OthersEI decreases with the increase of the issue width.

CRC32 (Table VI(d)). Only one kind of FirstEI is derived in CRC32, as shown in Table VI(d). FirstEI contains five operations out of which four operations are critical and one is contentious.

Relaxing the register file constraint would not improve the performance of all EIs. Since the hottest basic block in CRC32 takes up over 99% of the total execution cycles, only one block is selected to explore the EI. In this basic block, all profitable operations can be legally packed into the EIs under 4/2/2. Therefore, no further performance gain can be achieved.

Increasing the issue width would reduce the performance of Pro-FirstEI and Pro-OthersEI. The reason is similar to that of Blowfish.

Dijkstra (Table VI(e)). Only two EIs are derived in Dijkstra for all constraints. One is FirstEI, as shown in Table VI(e). The other contains two critical operations, which means that its structure does not change with changes in constraint.

As was observed with CRC32, relaxing the register file constraint does not improve the performance for Pro-FirstEI and Pro-OthersEI, since all profitable operations have been packed into EIs under 4/2/2.

Increasing the issue width would result in a small performance gain for Pro-FirstEI and Pro-OthersEI. Since increasing the issue width would reduce the total number of execution cycles (without EI), it may provide further performance gains.

Rijndael (Table VI(g)). The size of FirstEI derived in Rijndael is highly dependent on the number of register read/write ports. Since four register file constraints are used in this experiment, four kinds of FirstEI are derived, and all are shown in Table VI(g). As was the case with Blowfish, many EIs derived in Rijndael can be merged with FirstEI.

Relaxing the register file constraint can improve the performance of Pro-FirstEI. Among all the benchmarks selected, Rijndael has the largest number of profitable operations. Even if the register file constraint is 10/5, not all reachable and profitable operations can be packed into a single EI. Thus, relaxing the register file constraint can let the EI legally contain more operations and achieve a higher performance gain. On the other hand, relaxing the register file constraint decreases the performance of Pro-OthersEI. This is because many profitable operations have been packed into FirstEI.

Increasing the issue width would decrease the performance of Pro-FirstEI and Pro-OthersEI. This is because Rijndael has a very high ILP measure of about 5.62/65 (avg./max.), and thus, its performance would benefit more from an increased issue width than from the EI. Therefore, for an application with a very high ILP, it is difficult to improve the performance by increasing the issue width.

Stringsearch (Table VI(f)). Only one kind of FirstEI is derived in Stringsearch, as shown in Table VI(f). Since the hottest basic block where FirstEI is derived takes up over 95% of the total execution cycles, EIs derived in the hottest basic block dominate the total speedup.

Relaxing the register file constraint would not improve the performance of Pro-FirstEI. This is because all profitable operations have been legally packed into FirstEI under 4/2/2. For Pro-OthersEI, relaxing the register file constraint would improve the performance, except for the case where 8/4 changes to 10/5. This is because all profitable operations have been packed into EI when the register file constraint is 8/4. Therefore, no further performance gains are possible when the register file constraint is higher than 8/4.

Increasing the issue width would improve the performance of Pro-FirstEI but not Pro-OthersEI. Since increasing the issue width would reduce the total number of execution cycles (without EI), the performance of Pro-FirstEI would therefore improve. On the other hand, increasing the issue width would decrease the number of profitable operations for OthersEI, resulting in a negative speedup.

5. CONCLUSION

This article discusses two important factors that need to be considered when exploring EI in multiple-issue architectures. First, packing critical (located on critical paths) and contentious (operations that might have a delay in execution due to hardware resource contention) operations achieves the higher speedup. Second, the performance

impact of each operation on the total schedule may change dynamically when critical and/or contentious operations are replaced by EI(s). Based on these considerations, an EI exploration algorithm for multiple-issue architectures is proposed and compared against the approach from a previous study. We also evaluated four variations of our algorithm to study the speedup, area cost, and area efficiency. The experimental results show that accounting for these two factors (criticalness and contentiousness) yields a higher performance gain.

We have also investigated the impact of relaxing the register file constraints and increasing the issue width. Relaxing the register file constraints usually increases the performance of EI, because more operations can be packed into one EI. Increasing the issue width does not always improve the performance. For the case of the bottleneck being the critical path of the schedule after increasing the issue width, adapting EI to reduce the critical path would result in better speedup. Otherwise, the performance would reduce when the issue width increases.

Among the many topics which may be explored in future research, two important ones are described as follows. First, single-instruction multiple data (SIMD) is a widely used EI that improves the execution performance of both single- and multiple-issue architectures. However, in most SIMD ISA, each instruction only executes a simple arithmetic/logical operation, such as addition or subtraction. To enhance the capability of SIMD instructions, it would be interesting to explore the packing of multiple data-independent and identical EIs into a single SIMD instruction extension. Second, since the area cost is an important design consideration in some embedded systems, reducing this cost would be an interesting topic. According to experimental results, the area cost highly depends on the similarity among the derived EIs (i.e., if the derived EIs are as similar as possible, more area cost could be saved). Consequently, if the algorithm can take the similarity into account when exploring EIs, the area cost can be reduced.

REFERENCES

ALTERA CORP. 2004. *Nios II Processor Reference Handbook.* http://www.altera.com/literature/lit-nio2.jsp.

K. Atasu, L. Pozzi, and P. Ienne. 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference* (*DAC*). 256–261.

P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. 2006. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. *IEEE Trans. Integ. VLSI Syst.* 14, 7, 754–762.

N. T. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, and V. Nieuwenhove. 2004. OptimoDE: Programmable accelerator engines through retargetable customization. In *Proceedings of the Symposium on High Performance Chips* (*HotChips*).

N. T. Clark, H. Zhong, and S. A. Mahlk. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.* 54, 10, 1258–1270.

G. David, M. A. Ertl, and A. Krall. 2001. A fast Java interpreter. In *Proceedings of the Java Optimization Strategies for Embedded Systems Workshop* (*JOSES*).

P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. 2000. LX: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (*ISCA*). 203–213.

C. Galuzzi and K. Bertels. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfi. Technol. Syst.* 18.

D. Goodwin and D. Petkov. 2003. Automatic generation of application specific processors. In *Proceedings of the International Conference on Compilers Architectures and Synthesis for Embedded Systems* (*CASES*). 137–147.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brow. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization* (*WWC*). 3–14.

T. R. Halfhill. 2000. ARC cores encourages plug-ins. *Microprocess. Rep.* 14, 4, 42–44.

T. R. Halfhill. 2003a. MIPS embraces configurable technology. *Microprocess. Rep.*

T. R. Halfhill. 2003b. Tensilica's software makes hardware. *Microprocess. Rep.*

D. Jain, A. Kumar, L. Pozzi, and P. Ienne. 2004. Automatically customising VLIW architectures with coarse grained application-specific functional units. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems* (*SCOPES*). 17–32.

C. Lattner. 2002. LLVM: An infrastructure for multi-stage optimization. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, IL.

C. Liem, T. May, and P. Paulin. 1994. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the European Design and Test Conference* (*ED&TC*). 31–37.

A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. 2003. A VLIW processor with reconfigurable instruction set for embedded applications. *IEEE J. Solid-State Circuits*. 38, 11, 1876–1886.

Y. S. Lü, L. Shen, L. Huang, Z. Y. Wang, and N. Xiao. 2008. Customizing computation accelerators for extensible multi-issue processors with effective optimization techniques. In *Proceedings of the 45th Annual Design Automation Conference* (*DAC*). 197–200.

L. Pozzi and P. Ienne. 2006. Automatic instruction set extension. In *Customizable Embedded Processors*, Morgan Kaufmann, San Mateo, CA.

L. Pozzi, K. Atasu, and P. Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. Comput.-Aid. Des. Integ. Circuits Syst.* 25, 7, 1209–1229.

D. S. Rao and F. J. Kurdahi. 1992. Partitioning by regularity extraction. In *Proceedings of the 29th Design Automation Conference* (*DAC*). 235–238.

V. S. Reddy. 2006. Exploring VLIW ASIP design space using trimaran based framework. Master's thesis. Department of Computer Science and Engineering, Indian Institute of Technology Delhi.

M. A. R. Saghir, M. El-Majzoub, and P. Alk. 2007. Customizing the datapath and ISA of soft VLIW processors. In *Proceedings of the 2nd International Conference an High Performance Embedded Architectures and Compilers* (*HiPEAC*). Lecture Notes in Computer Science, vol. 4367, Spring, 276–290.

W. Stephan, T. V. As, and G. Brown. 2008. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *Proceedings of the IEEE International Conference on Field-Programmable Technologies* (*ICFPT*). 369–372.

F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. 2002. Synthesis of custom processors based on extensible platforms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*). 641–648.

I. W. Wu, S.-C. Huang, C.-P. Chung, and J.-J. Shann. 2007. Instruction set extension generation with considering physical constraints. In *Proceedings of the 2nd International Conference on High Performance Embedded Architecture and Compilers*. Lecture Notes in Computer Science, vol. 4367, Springer-Verlag, Berlin Heidelberg, 291–305.

P. Yu and T. Mitra. 2007. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the International Conference on Field Programmable Logic and Applications* (*FPL*). 273–278.

Received September 2012; revised July 2013; accepted September 2013